Practical use of static composition of refactoring operations

Julien Cohen, Akram Ajouli

Practical use of static composition of refactoring operations

Julien Cohen
ASCOLA team (EMN, INRIA, LINA)
University of Nantes
Julien.Cohen@univ-nantes.fr

Akram Ajouli
ASCOLA team (EMN, INRIA, LINA)
École des Mines de Nantes
Akram.Ajouli@mines-nantes.fr

ABSTRACT

Refactoring tools are commonly used for remodularization tasks. Basic refactoring operations are combined to perform complex program transformations, but the resulting composed operations are rarely reused, even partially, because popular tools have few support for composition. In this paper, we recast two calculus for static composition of refactorings in a type system and we discuss their use for inferring useful properties. We illustrate the value of support for static composition in refactoring tools with a complex remodularization use case: a round-trip transformation between programs conforming to the Composite and Visitor patterns.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Design, Theory, Experimentation

Keywords

Refactoring, Remodularization, Modularity, Program Transformations, Static Analysis, Type Systems

1. INTRODUCTION

Most development environments now provide automatic refactoring operations to support remodularization. A few basic operations, such as *rename* or *safe delete*, are popular because they are reliable. On the other side, more complex operations are less used because they are unstable. Indeed, building complex refactoring operations is hard. Complex operations are often built by combining several base operations, but development environments do not provide support to ensure that such combinations are coherent, and the resulting operations are not more stable than monolithic ones.

Automatically checking the properties of combinations of operations would help for the design, the validation and the maintenance of reliable, complex refactorings. Consider for instance the four following properties:

Non-Failure: *The operation ends without failure.* A basic operation fails when one of its preconditions is not satisfied. A composed operation fails when the precondition of one of its components is not satisfied by the program resulting from the previous operation.

Behavior Preservation: *The operation does not change the external behavior of the program.* This is ensured for instance when the preconditions of each base operation are strong enough to ensure that these operations preserve the behavior.

Correctness of the Effect: *The resulting program has the expected architecture* (when the operation succeeds).

Round-trip Transformation: *Two operations provide inverse effects.* Such pairs of operations can be used to navigate between different architectures for a same program [3].

We are interested in checking these properties statically on composed operations (without performing the refactoring) in order to help building and validating complex transformations. Some authors provide systems to compute different properties of composed operations [11, 9]. The system of Ó Cinnéide and Nixon [11] generates convenient post-conditions from specified preconditions, while the system of Kniesel and Koch [9] does the opposite. At first sight, it is not clear whether these systems are equivalently powerful, and which system to use to check the properties given above.

In this paper, we combine these two systems in a single calculus, formulated as a static type system to show how they are related to each other (Sec. 2). Then we show how the properties given above are handled in that system (Sec. 3). Finally, we present a case study to validate the system and to illustrate its use to check the given properties: a *Composite* $\leftrightarrow$ *Visitor* transformation in Java, based on the refactoring capabilities of the popular IDE IntelliJ IDEA (Sec. 4).

2. A TYPE SYSTEM FOR COMPOSITION

We reformulate the two considered composition systems in an unified formalization. We use the vocabulary and notations of type systems for suitable concepts and mechanisms.

For instance, the type $P \to Q$ describes refactoring operations that produce programs with the property Q when applied to programs with the property P.

2.1 Refactoring Operations

Formulas

Refactoring operations usually have preconditions. In refactoring tools, the preconditions are implemented by a static analysis of the subject program. We abstract these analyses by atomic propositions such as ExistsClass("Foo") (all the examples hold for Java), which are also used to denote sets of programs (for the example, the set of programs which contain a declaration of a class named Foo, or which import such a class).

To describe preconditions, we consider formulas of propositional logic on these atomic propositions, with usual connectors for negation, conjunction and disjunction. For instance, ExistsClass("Foo")$\wedge$ ExistsMethod("Foo","bar") is a formula. We note $\Rightarrow$ the logic implication connector.

Base Refactoring Operations

We consider a set $\mathbb{R}$ of base refactoring operations, such as AddClass("Foo"). Each base refactoring operation comes with a precondition and two functions that describe its behavior. We reference these preconditions and behavior functions by the means of a total function $\mathcal{R}$ on $\mathbb{R}$ such that, for any base operation t, $\mathcal{R}(t) = (P, f, b)$ means that:

- The formula P is the precondition of t.
- The function f, called the *forward description* of t (generalizing *postcondition mappings* of [11]), describes the effect of the operation on programs: $f(Q) = R$ means that if the formulas Q and P hold before applying t then R will hold after. For instance, for RenameClass(A, B), we have $f(\mathsf{ExistsMethod}(A, m)) = \mathsf{ExistsMethod}(B, m)$.
- The function b, called the *backward description* of t (we consider *backward descriptions* of [9] as functions), also describes the effect of t, in a different way: $b(R) = Q$ means that Q and P have to hold before applying t to ensure that R holds after. For instance, for RenameClass(A, B), we have $b(\mathsf{ExistsMethod}(B, m)) = \mathsf{ExistsMethod}(A, m)$ and $b(\mathsf{ExistsMethod}(A, m)) = \bot$.

Of course, these descriptions of the refactoring operations must be faithful to the underlying tool.

The functions f and b are usually not injective and therefore they are not invertible. Furthermore, it is possible to provide only forward descriptions or only backward descriptions in $\mathcal{R}$ (with the consequences explained later).

Soundness of the Operations

We say that a set of refactoring operations $\mathbb{R}$ with its description $\mathcal{R}$ is *sound* if its operations preserve the behavior when their preconditions are satisfied.

Transformations/Refactoring Chains

A *transformation* based on a set of refactoring operations $\mathbb{R}$ (or a *refactoring chain*) is a chain of base refactoring operations of $\mathbb{R}$. The set of transformations is defined by the following grammar where t denotes a refactoring operation ($t \in \mathbb{R}$, the semicolon is associative):

$$T ::= t \mid T; T$$

Applying a transformation $t_1; t_2; ...; t_n$ to a program consists in applying t_1 if its precondition holds, and then applying t_2 on the result of the application of t_1, if the precondition of t_2 holds on that new program, and so on. If one of the preconditions fails on the corresponding intermediate program, the transformation *fails* on the initial program.

2.2 The Type System

Typing Judgments

For a transformation T based on a set of operations $(\mathbb{R}, \mathcal{R})$, we write $\mathcal{R} \vdash T : P \to Q$ when the transformation T does not fail on any program satisfying P and provides a program satisfying Q.

As a particular case, we write $\mathcal{R} \vdash T : P \to \top$ when the transformation T does not fail on any program satisfying P ($\top$ is the *true* formula).

Typing Rules

The following rules are used to prove that type judgments hold.

$$\frac{\mathcal{R}(t) = (P, f, b) \quad Q \Rightarrow P}{\mathcal{R} \vdash t : Q \to f(Q)} \quad (\textit{forward-description})$$

$$\frac{\mathcal{R}(t) = (P, f, b)}{\mathcal{R} \vdash t : (b(Q) \wedge P) \to Q} \quad (\textit{backward-description})$$

$$\frac{P \Rightarrow P' \quad \mathcal{R} \vdash T : P' \to Q' \quad Q' \Rightarrow Q}{\mathcal{R} \vdash T : P \to Q} \quad (\textit{weakening})$$

$$\frac{\mathcal{R} \vdash T_1 : P \to Q \quad \mathcal{R} \vdash T_2 : Q \to R}{\mathcal{R} \vdash (T_1; T_2) : P \to R} \quad (\textit{sequence})$$

The *forward/backward-description* rules directly reflect the calculus of [11] and [9]. The *weakening* and *sequence* rules are similar to corresponding rules in Hoare logic [7] (a transformation boils down to a sequence of actions that change the state of a source code). The *weakening* rule also expresses the usual contravariance of types on the left hand side of the arrow and covariance on the right hand side. This kind of subtyping comes from the inclusion of sets of programs denoted by ordered formulas.

The proofs of implications in the premise of the rules *weakening* and *forward-inference* can be led by external solvers for propositional logic.

Type Inference

Typing rules allow to check that a transformation has a given type. Now, we are interested in generating (most general) types for transformations.

Given a (pre)condition P and a refactoring chain T, *forward inference* consists in applying the *sequence* and *forward-description* rules to compute a Q so that $\mathcal{R} \vdash T : P \to Q$ holds.

Inversely, given a (post)condition Q , *backward inference* consists applying the *sequence* and *backward-description* rules to compute a P so that $\mathcal{R} \vdash T : P \to Q$ holds.

This is the main difference in the use of the two calculi.

Finally, to prove $\mathcal{R} \vdash T : P \to Q$ for given P and Q, you can use forward inference from P to find a postcondition Q' and use the *weakening* rule with $Q' \Rightarrow Q$, or you can use backward inference from Q to compute a precondition P' and use the *weakening* rule with $P \Rightarrow P'$.

Note that backward inference will generate the weakest precondition you can prove with the given forward-descriptions, and inversely for forward inference.

3. TRANSFORMATION PROPERTIES: PRACTICAL USE

Non-Failure

Non-Failure is established by proving the type judgment $\mathcal{R} \vdash T : P \to \top$.

If you want to check Non-Failure for a given precondition P, you can either use forward inference and then the weakening rule with the premise $\top \Rightarrow Q$ to check that the computed postcondition Q is not empty, or use the backward inference from $\top$ and the weakening rule with the premise $P \Rightarrow P'$ to compare the computed precondition P' to P.

If you don't have a precondition to start with, backward inference allows to compute a convenient one. If you don't have backward descriptions, you will still be able to prove Non-Failure but you have to "guess" a convenient precondition.

Behavior Preservation

To ensure behavior preservation of refactoring chains, it is sufficient to use only behavior preserving base operations. However, refactoring operations of popular IDEs do not always preserve the semantics. The solution is to use more restrictive preconditions for these operations to ensure that they are always used in a context where they preserve the behavior. The refactoring tool does not need to be modified, it is sufficient to consider these restricted preconditions in $\mathcal{R}$ (soundness of $\mathcal{R}$).

Correctness of the Effect

To ensure the Correctness of the Effect, you have to specify the expected target architecture with a formula Q, and to prove the assertion $\mathcal{R} \vdash T : P \to Q$ for a non empty P. Again, If you have backward descriptions in $\mathcal{R}$, a convenient precondition P can be computed by backward inference from the specified postcondition Q.

Also, with backward *and* forward inference, you can compute a weakest precondition P such that $T : P \to Q$ and then compute the strongest postcondition Q' from P such that $T : P \to Q'$ to have a more precise specification of the reached architecture (if forward descriptions are precise enough to compute a Q' stronger than Q, i.e. $Q' \Rightarrow Q$).

Round-Trip Transformation

A pair of transformations (T_1, T_2) provides a Round-Trip Transformation between two sets of programs denoted by the formulas R_1 and R_2 when the two following assertions hold:

$$\mathcal{R} \vdash T_1 : R_1 \to R_2$$

$$\mathcal{R} \vdash T_2 : R_2 \to R_1$$

Such formulas R_1 and R_2 can be generated by applying backward *or* forward inference with a fix-point strategy. Again, initial pre/postconditions possibly imposed are dealt with the *weakening rule* and can be used as starting point for the fix-point search.

Design and Maintenance of Transformations

Type inference, as for most static type systems, helps to design correct transformations and is complementary to dynamic testing. Also, since the inferred types are the most general possible (with respect to a given description of the refactoring operations), type inference allows to discover that a transformation is more general than expected. Types can also be used for documentation.

In the following section, we use our type system to validate a complex round-trip transformation.

4. CASE STUDY: COMPOSITE $\leftrightarrow$ VISITOR TRANSFORMATION

As a case study, we consider a Composite $\leftrightarrow$ Visitor transformation. Because the Composite and Visitor design patterns have dual properties with respect to modularity, this kind of transformation illustrates a complex remodularization with deep changes in the micro-architecture. Also, such transformations address the tyranny of the dominant decomposition [3], a central problem for separation of concerns. Such a Visitor$\to$Composite transformation has been implemented on a real interpreter [6]. In this section, we consider the implementation of [1], based on the refactoring tool of JetBrains IntelliJ IDEA (free edition, version 11.0.2).

In [6] and [1], the transformations are validated by testing. Here, using the composition calculus to assess the properties discussed above provides an additional validation and a better understanding of the round-trip transformation. That use case also shows the type system and the practical properties in action.

4.1 Implementation

The difficult part of the implementation is not the typing rules, it is the definition of a function $\mathcal{R}$ which describes faithfully the underlying refactoring tool. Our description of the base refactoring operations (preconditions and backward descriptions) is given in our research report [2].

In this experiment, some aspects of the language are not taken into account in $\mathcal{R}$, such as method visibility.

In the rest of this section, $\mathcal{R}$ is left implied.

4.2 Composite $\to$ Visitor Transformation

We consider the Composite $\to$ Visitor algorithm of Fig. 2, which is designed to be applied to recursive class hierarchies (with composites) according to the Composite or Interpreter pattern. That transformation has been successfully tested on the program of Fig. 1 (the result program is given in [2]).

In that algorithm, S is the root class of the composite hierarchy, LC is the list of classes in that hierarchy, LM is the list of business methods, *aux* is a function to generate fresh intermediate names, and V is a function to generate visitor class names from method names.

Non-Failure

We first compute the minimum precondition that ensures non-failure. That precondition is noted $P_\top$ (Fig. 3). Thus

```
abstract class Graphic {

    abstract public void show() ;

    abstract public void fullprint() ;
}
```

```
class Square extends Graphic{

  public int side;

  public void show() {
    System.out.print("SQ :" + side);
  }

  public void fullprint(){
    System.out.println("square " + this + ".");
  }
}
```

```
class Container extends Graphic {

  public ArrayList<Graphic> childs =
    new ArrayList<Graphic> ();

  public void show() {
    System.out.print("CT:");
    for (Graphic g : childs) g.show();
  }

  public void fullprint(){
    System.out.print("container " + this + " with:");
    for (Graphic g : childs) g.fullprint();
    System.out.println("(end)");
  }
}
```

Figure 1: Tested initial program (*subject* program)

CompositeToVisitor(S,LC,LM,aux,V) =

1. ForAll m in LM do
   CreateEmptyClass(V(m))
2. ForAll m in LM do
   CreateIndirectionInSuperClass(S,m, aux(m))
3. ForAll m in LM, c in LC do
   InlineMethodInvocations(c, m, aux(m))
4. ForAll m in LM do
   AddParameterWithReuse(S, aux(m), V(m),
   "new {V(m)}()")
5. ForAll m in LM, c in LC do
   MoveMethodWithDelegate(c, aux(m), V(m), "visit")
6. ExtractSuperClass(LV, "Visitor")
7. ForAll m in LM do
   GeneraliseParameter(S, aux(m), V(m), "Visitor")
8. Let LAUX = { aux(m) }$_{\mathsf{m}\in\mathsf{LM}}$ in
   MergeDuplicateMethods(S, LAUX, "accept")

Figure 2: Composite→Visitor transformation [1]

the *Composite*→*Visitor* chain has the type $P_\top \to \top$.

From that precondition, we learn for instance that the special identifier *this* must not be involved in an overloading static resolution in the recursive business methods, which is not stated by Ajouli [1]. The witness program of Fig. 1 satisfies that precondition $P_\top$.

$\neg$ExistsMethodDefinition($Graphic, accept$)
$\wedge$ $\neg$ExistsMethodDefinition($Square, accept$)
$\wedge$ $\neg$ExistsMethodDefinition($Container, accept$)
$\wedge$ $\neg$IsInheritedMethod($Graphic, accept$)
$\wedge$ NotInvolvedInOverloading($Container, show, this$)
$\wedge$ IsLocalInvokedMethod($Square, show, this, Graphic$)
$\wedge$ NotInvolvedInOverloading($Square, show, this$)
$\wedge$ $\neg$ExistsType($Visitor$)
$\wedge$ ExistsClass($Square$)
$\wedge$ $\neg$BoundVariableInMethodBody($Graphic, show, v$)
$\wedge$ $\neg$BoundVariableInMethodBody($Graphic, fullprint, v$)
$\wedge$ IsRecursiveMethod($Container, show$)
$\wedge$ ExistsClass($Container$)
$\wedge$ IsRecursiveMethod($Container, fullprint$)
$\wedge$ ExistsMethodDefWithParams($Graphic, show, []$)
$\wedge$ ExistsAbstractMethod($Graphic, show$)
$\wedge$ $\neg$IsInheritedMethod($Graphic, showTmp$)
$\wedge$ $\neg$IsInheritedMethodWithParams($Graphic, showTmp, []$)
$\wedge$ $\neg$ExistsMethodDefWithParams($Graphic, showTmp, []$)
$\wedge$ HasReturnType($Graphic, show, void$)
$\wedge$ ExistsMethodDefinition($Graphic, show$)
$\wedge$ ExistsMethodDefinition($Square, show$)
$\wedge$ ExistsMethodDefinition($Container, show$)
$\wedge$ $\neg$ExistsMethodDefinition($Graphic, showTmp$)
$\wedge$ $\neg$ExistsMethodDefinition($Square, showTmp$)
$\wedge$ $\neg$ExistsMethodDefinition($Container, showTmp$)
$\wedge$ $\neg$IsOverloaded($Graphic, show$)
$\wedge$ $\neg$IsOverloaded($Square, show$)
$\wedge$ $\neg$IsOverloaded($Container, show$)
$\wedge$ ExistsClass($Graphic$)
$\wedge$ IsAbstractClass($Graphic$)
$\wedge$ ExistsMethodDefWithParams($Graphic, fullprint, []$)
$\wedge$ ExistsAbstractMethod($Graphic, fullprint$)
$\wedge$ $\neg$IsInheritedMethod($Graphic, fullprintTmp$)
$\wedge$ $\neg$IsInheritedMethodWithParams($Graphic, fullprintTmp, []$)
$\wedge$ $\neg$ExistsMethodDefWithParams($Graphic, fullprintTmp, []$)
$\wedge$ AllSubclasses($Graphic, [Square; Container]$)
$\wedge$ HasReturnType($Graphic, fullprint, void$)
$\wedge$ ExistsMethodDefinition($Graphic, fullprint$)
$\wedge$ ExistsMethodDefinition($Square, fullprint$)
$\wedge$ ExistsMethodDefinition($Container, fullprint$)
$\wedge$ $\neg$ExistsMethodDefinition($Graphic, fullprintTmp$)
$\wedge$ $\neg$ExistsMethodDefinition($Square, fullprintTmp$)
$\wedge$ $\neg$ExistsMethodDefinition($Container, fullprintTmp$)
$\wedge$ $\neg$IsOverloaded($Graphic, fullprint$)
$\wedge$ $\neg$IsOverloaded($Square, fullprint$)
$\wedge$ $\neg$IsOverloaded($Container, fullprint$)
$\wedge$ $\neg$ExistsType($ShowVisitor$)
$\wedge$ $\neg$ExistsType($FullprintVisitor$)

Figure 3: Computed precondition for Non-Failure (formula $P_\top$) instantiated with method names and class names of the program of Fig. 1

Correctness of the Effect

We now consider the correctness of the effect. We suppose the target Visitor micro-architecture is described by the formula Q_V of Fig. 4, which we take as a postcondition.

That specification describes coarsely the target Visitor architecture (accurate description of the Visitor pattern is out of scope of this paper).

Here, the computed precondition is the same as with *true*

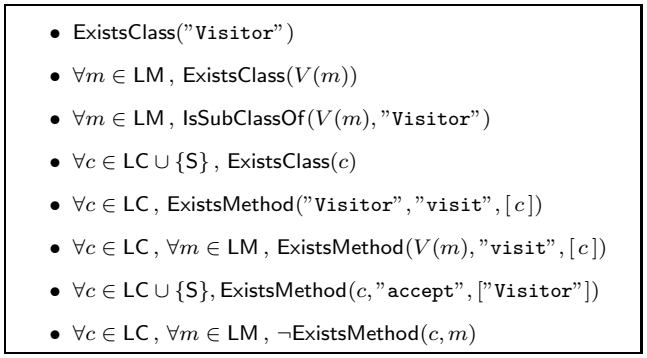

Figure 4: Weak specification of the target Visitor architecture (formula Q_V)

as a postcondition. Thus the chain has also the type $P_\top \to Q_V$, which is more precise than $P_\top \to \top$. Note that forward inference may allow to find a more precise postcondition Q so that $P_\top \to Q$ is still a type for that chain (we did not implement forward descriptions in this case study).

We next address the round trip transformation (the return transformation is described in [2]).

4.3 Round-Trip Transformation

We now consider the chain composed of *Composite* $\to$ *Visitor* and *Visitor* $\to$ *Composite*. We call T_{CVC} that chain. We are looking for a formula P_X such that $T_{CVC} : P_X \to P_X$ (the fix-point for T_{CVC}).

We find that fix-point with backward inference and the following strategy: we find a P_0 such that $T_{CVC} : P_0 \to \top$, then a P_1 such that $T_{CVC} : P_1 \to P_0$, and so on until $P_i = P_{i+1}$ (this is equivalent to finding a fix-point for two formulas as explained in Sec. 3)

In fact, in this use case, the fix-point P_X is obtained at the first iteration, for P_0, the precondition for Non-Failure. That precondition is sufficient to ensure itself as a postcondition. However, this is not representative of the general case.

The formula P_X is very close to the formula $P_\top$ for Non-Failure of *Composite* $\to$ *Visitor* (Fig. 3). The only differences are proposition that ensure that the temporary method names for the return transformation do not create a clash.

4.4 Results

Validation of the Transformation

We have formally validated the transformation: non failure, correctness, round-trip transformation and behavior preservation. However, this validation must be taken with care because the faithfulness and soundness of the axioms used (the implemented $\mathcal{R}$) are not formally validated themselves (see the discussion in Sec.6).

Domain of the transformation

The computed precondition for the round-trip transformation allows to define precisely the set of acceptable input programs.

This is useful to know if that transformation can be reused for some other programs than the initial one, or to know when the transformation has to be adapted to cope with a modification of the subject program during its maintenance lifetime.

Accuracy of Backward Descriptions

In order to reach convenient results, we have had to give very accurate preconditions and backward descriptions for some operations. For instance, for the *DeleteClass* operation (we delete visitor classes in the Visitor$\to$Composite chain), a predicate *NotIsUsedClass* is too coarse, because no operation has it as a postcondition. In this case, we had to use several predicates that tell that the type for the class is not used (for variables, parameters, return types or extensions) and that its constructors, methods and variables are not used. In such situations, the work to ensure that the composition succeeds can be compared to the use of formal proof assistants which force you to give all the justifications to ensure that a property holds.

This is the classic dilemma in the design of type systems between coarse types which are easy to handle but which lead to rejection of some correct programs and very accurate types, more difficult to prove but which reject less programs.

Validation of the Composition System

This use case also validates the practical use of the composition system (here, only backward inference is used however) with the retained properties: Non-Failure, Correctness of the Effect, Round-Trip Transformation. Is also shows that the constraint of using only behavior preserving operations in order to ensure the behavior preservation for the whole chain is acceptable, even for complex remodularizations.

5. RELATED WORK

Ó Cinnéide and Nixon [11] and Kerievsky [8] give many examples of remodularization towards design patterns by chaining refactoring operations. Hills et al. [6] show that it can be successful for drastic architecture changes in real programs (they transform a real interpreter based on the Interpreter Pattern to the Visitor pattern). To our knowledge, our use case is the largest transformation to be formalized.

There are many contributions to the domain of static composition of refactorings. For instance it has been much developed by Roberts [13]. We have selected Ó Cinnéide and Nixon [11] and Kniesel and Koch [9] for our work because they are representative of the two ways of inference. In order to unify these two systems, we have taken only their basic features (also for simplicity of the presentation and lack of space). For instance, in [9], the transformations are program independent. We have not done that, but it should be straightforward in our system.

Several frameworks for composition of refactorings are currently available. For instance JTransformer/StarTransformer[1] (ROOTS group, Univ. of Bonn) is under active development. We did not use it for our use case because we wanted to deal with the refactoring operations for which the transformation was designed for, and because it does not support pre/postcondition inference.

6. CONCLUSION

Contributions

This paper reformulates two existing composition calculus into a type system, shows their practical use, and illustrates

[1] `http://sewiki.iai.uni-bonn.de/research/jtransformer/start`

it on a non-trivial use case of remodularization. More precisely:

- We have recast two composition systems, one based on forward descriptions and one based on backward descriptions, into one calculus. This allows to make them interact. For instance, for a given transformation, one can first compute a minimal precondition for an expected target architecture (backward inference), then compute a more precise characterization of the resulting programs with that precondition (forward inference). This also makes them comparable. In particular, they both allow type checking, but they are complementary for type inference.
- The resulting system is language- and tool- independent because predicates on programs and descriptions of refactoring operations are parameters of the system.
- We have shown how to use the resulting system in various situations with four general, yet meaningful properties on transformations, considering that initial preconditions and postconditions are specified or not.
- We have applied our system to check these properties on a deep remodularization (50 operations). That use case shows that the system is workable with complex transformations and it illustrates its value (validation of the transformation, identification of the acceptable inputs).

However, static composition of refactorings still has the following drawbacks:

- Defining a faithful and sound $\mathcal{R}$ for the refactoring tool at hand is hard. We can rely on proofs of correctness for some operations, for instance [4] and [14] for Java, but many operations are not proven yet.
- Defining $\mathcal{R}$ with the convenient accurateness for the transformation at hand is also hard.

Future Work

To make static composition adopted by a large set of programmers, it should be integrated in a popular IDE, soundly based on the native refactoring operations of that IDE. Here are a few other possible future works:

- Compare more deeply forward and backward descriptions: can one be inferred from the other?
- Consider transformations with loops or conditional branches. Again, we can rely on Hoare logic. For instance, the following rule for conditionals fits well:

$$\frac{\mathcal{R} \vdash T_1 : (P \wedge Q) \rightarrow R \quad \mathcal{R} \vdash T_2 : (P \wedge \neg Q) \rightarrow R}{\mathcal{R} \vdash \mathit{if}\ Q\ \mathit{then}\ T_1\ \mathit{else}\ T_2 : P \rightarrow R}$$

- Study how the type system helps to make evolve a transformation when the architecture of the subject program has been modified.
- Use types to reduce the space of search in inference of refactoring chains [10, 12, 5].

We hope this work will contribute to a better integration of composition of refactorings in IDEs.

Acknowledgments

The authors would like to thank Anna Kozlova (JetBrains), Günter Kniesel (Univ. of Bonn), Mel Ó Cinnéide (Univ. College Dublin) and Jean-Claude Royer (École des Mines de Nantes) for their comments on this work.

References

[1] A. Ajouli. An automatic reversible transformation from composite to visitor in Java. In *Conférence en IngénieriE du Logiciel (CIEL)*, 2012. 6 pages.

[2] A. Ajouli and J. Cohen. Refactoring Composite to Visitor and Inverse Transformation in Java. Research report hal-00652872 (version 2), 2011/2012. URL `http://hal.archives-ouvertes.fr/hal-00652872/en`.

[3] J. Cohen, R. Douence, and A. Ajouli. Invertible program restructurings for continuing modular maintenance. In *Soft. Maintenance and Reengineering (CSMR), European Conf. on*, pages 347–352, 2012.

[4] A. Garrido and J. Meseguer. Formal specification and verification of java refactorings. In *IEEE Int. Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 165–174. IEEE, 2006.

[5] I. Hemati Moghadam and M. Ó Cinnéide. Automated refactoring using design differencing. In *Soft. Maintenance and Reengineering (CSMR), European Conf. on*, pages 43–52, 2012.

[6] M. Hills, P. Klint, T. Van Der Storm, and J. Vinju. A case of visitor versus interpreter pattern. In *Int. Conf. on Objects, Models, Components, Patterns (TOOLS)*, pages 228–243. Springer-Verlag, 2011.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, Oct. 1969.

[8] J. Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004.

[9] G. Kniesel and H. Koch. Static composition of refactorings. *Science of Computer Programming*, 52(Issues 1-3): 9–51, 2004. Special Issue on Program Transformation.

[10] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 6(3):269–285, 2007.

[11] M. Ó Cinnéide and P. Nixon. Composite refactorings for Java programs. In *Workshop on Formal Techniques for Java Programs, ECOOP*, 2000.

[12] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *IEEE Int. Conf. on Soft. Maintenance (ICSM)*, 2010.

[13] D. B. Roberts. *Practical analysis for refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1999.

[14] M. Schäfer, J. Dolby, M. Sridharan, E. Torlak, and F. Tip. Correct refactoring of concurrent java code. In *European Conf. on Object-Oriented Programming*, ECOOP, pages 225–249. Springer-Verlag, 2010.